# Topics in Computational Inference

David M. Allen
University of Kentucky

November 4, 2018

# 12 Calling *C* and *C++* from *R*

The definitive reference for calling *C* or *C++* from *R* is Chapter 5 of Writing *R* Extensions. The *.C* method is presented here. This method is also favored by Eubank and Kupresanin [1] and Matloff [2].

*Rcpp* is a more elaborate method, and is discussed in Rcpp: Seamless *R* and *C++* Integration.

# Section 12.1 — Introduction

*R* is a wonderful programming language. It can leap long matrix expressions in a single bound. However *R* can be slow for situations where frequent access to individual elements of a matrix and loops are required. The *C* and *C++* languages are often more efficient in these situations. A good strategy is to program such functions in *C* or *C++* and call them from *R*. This note gives a recipe for doing that.

# A Distinction Between *C* and *C++*

The function names must be unique in a *C* program. This is a benefit when interfacing with *R*. *C++* allows function overloading, *ie.* multiple functions with the same name. This requires *name mangling* in order to have unique identifiers. Also, class methods don't behave exactly like ordinary functions. Therefore, I give a *C* example and a *C++* example.

# Two Situations

I consider two situations:

- Code is being developed from the beginning with the objective of interfacing it with *R*. Consider using *C*, as interfacing it is simpler.
- There is an existing *C*++ class that is well tested and you want to use it without modification.

# The Fibonacci Series

The Fibonacci series $f_0, f_1, \cdots$ is a series where $f_0 = 0, f_1 = 1$, and all subsequent terms are the sum of the two preceding terms. Calculation of the first $n$ terms involves a loop and is easily implemented in $C$.

# The *C* Code

The demonstration *C* function takes the desired number of terms as input and returns the series. The file that contains the function must have an extension `.c`. The function must return `void`, and the arguments must be pointers. The code for the Fibonacci series is in Listing 12.1.

Listing 12.1: Fibonacci.c

```c
#include <R.h>
// typically n would not be a pointer
void Fibonacci(int* n, int* f)
  {
  if(*n < 2)
    *n = -1;
  else
    {
    f[0] = 0;
    f[1] = 1;
    for(int i = 2; i<*n; i++)
      f[i] = f[i-1]+f[i-2];
    }
  return;
  }
```

# Compilation

The code is in Fibonacci.c and is compiled from the command line by

```
R CMD SHLIB Fibonacci.c
```

On Linux or Unix, this will produce a file `Fibonacci.so`. The "so" stands for *shared object*. On Windows this will produce a file, `Fibonacci.dll`. The "dll" stands for *dynamic linked library*.

# Calling the Function from *R*

The calling *R* program must have

```
dyn.load("Fibonacci.so")
```

before the function is called. The function is called by `.C( args )` where `args` is the name of the function enclosed in double quotes, followed by the function arguments cast to their types. The code is in Listing 12.2.

Listing 12.2: Fibonacci.R

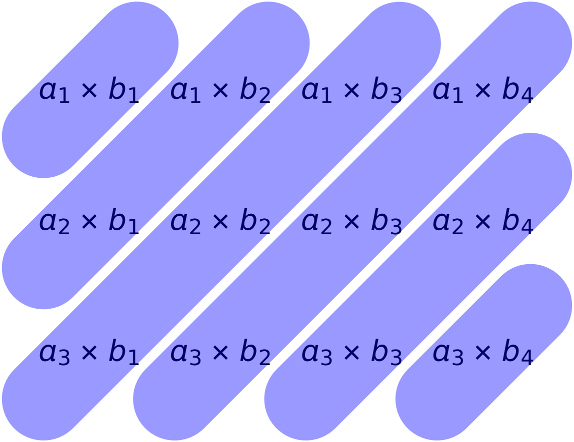
```
dyn.load("Fibonacci.so")

n      <- 10
# create memory for the result
f      <- vector(mode = "integer", length = n);
# f    <- rep(0, n)   # this would also work
result <- .C("Fibonacci", as.integer(n), as.integer(f))
# result contains a list with an element for
# each argument
print( result[[2]] )
```

# The Convolve Function

This example takes a *C++* class and *wraps it*, so that it looks like a group of *C* functions. The vehicle for illustration is the convolve function.

Convolve is the verb form of convolution. We might say we convolve vectors $a$ and $b$ to produce a convolution. In statistics the concept comes into play when finding the distribution of a sum of random variables. With reference to the next slide, the convolution of vectors $a$ and $b$ is a vector containing the sums within the blue bars.


$a_1 \times b_1$
$a_1 \times b_2$
$a_1 \times b_3$
$a_1 \times b_4$
$a_2 \times b_1$
$a_2 \times b_2$
$a_2 \times b_3$
$a_2 \times b_4$
$a_3 \times b_1$
$a_3 \times b_2$
$a_3 \times b_3$
$a_3 \times b_4$

# The Source Code

The class is such that constructor sets the size of the input vectors. Any number of vectors of that size can be processed with a single object of the class.

The header code is in Listing 12.3. The *C++* code is in Listing 12.4.

Listing 12.3: Convolve.h

```
#ifndef CONVOLVE
#define CONVOLVE

// design is such that evaluate could be called
// multiple times for same na & nb
class Convolve
  {
  int      na;
  int      nb;
  public:
           Convolve(int na, int nb);
          ~Convolve();
  void     evaluate(const double* a, const double* b,
                                             double* ab);
  };
#endif
```

Listing 12.4: Convolve.cpp

```
#include <cstring>  // memset is here
#include "Convolve.h"

Convolve::Convolve(int na, int nb)
  {
  this->na = na; this->nb = nb;
  }
Convolve::~Convolve(){}

void Convolve::evaluate(const double* a, const double* b,
                                              double* ab)
  {
  memset(ab, 0, (na + nb - 1)*sizeof(double));
  for(int i = 0; i < na; i++)
    for(int j = 0; j < nb; j++)
      ab[i + j] += a[i] * b[j];
  }
```

# Unit Testing

This is a brief diversion. Often when one builds a class, it is with the idea that it will be a module in a large system or program. However there should be a program that is designed to do nothing but test all the features of the class. This process of testing classes in isolation is called *unit testing*.

Code that exercises the Convolve class is in Listing 12.5.

Listing 12.5: driver.cpp

```
#include <cstdio>
#include "Convolve.cpp"

int main(int argc, char *argv[])
  {
  const int n = 6;
  double a[n];
  double b[n];

  for(int i = 0; i<n; i++)
    {
    a[i] = 1.0/n;
    b[i] = 1.0/n;
    }

  Convolve* con;
  double* ab = new double[n+n-1];
  con = new Convolve(n, n);
  con->evaluate(a, b, ab);
```

```
fprintf(stdout, "Distribution of sum\n");
fprintf(stdout, "%10s%10s\n", "value", "prob");
for(int i = 0; i<(n+n-1); i++)
  fprintf(stdout, "%10i%10.4lf\n", i+2, ab[i]);
return 0;
}
```

# Compiling the Driver

The following are instructions compiling and running the program using the *GNU Compiler Collection (GCC)*. If you use another compiler, see its instructions for compiling and running programs from source code. Open a terminal window and change to the directory containing the files. Issue the command

```
g++ driver.cpp -o driver
```

On Windows use `-o driver.exe`.

# Run the Driver

Run the program with the command

```
./driver
```

Windows users could omit the "`./`". Check the output with what it should be. Correct any errors and repeat.

# The *R* Interface File

For reasons mentioned earlier, *C*++ can't be directly loaded into *R*. The solution is to wrap *C*++ functions with *C* functions that follow conventions required by *R*. The interface or *wrapper* code for the current example is in Listing 12.6.

Listing 12.6: R-Convolve.cpp

```
#include <cstdio>
#include <cstdlib>
#include <R.h>
#include "Convolve.h"

static Convolve* conv;
extern "C"
  {

void setup(int* na, int* nb)  // arguments are pointers
  {
  conv = new Convolve(*na, *nb);
  }

void evaluate(double* a, double* b, double* ab)
  {
  conv->evaluate(a, b, ab);
  }
```

```
void closeout()
  {
  delete conv; // calls destructor
  }

  } // end extern "C"
```

# Compiling

Convolve.cpp and R-Convolve.cpp are compiled together from the command line by

```
R CMD SHLIB Convolve.cpp R-Convolve.cpp
```

This produces `Convolve.so` (or dll). The name is taken from the first file in the list.

# Using the Shared Object File

The code that loads and uses `Convolve.so` is

Listing 12.7: Convolve.R

```
dyn.load("Convolve.so")

lengths <- function(na, nb)
  {
  .C("setup", as.integer(length(a)),
                              as.integer(length(b)))
  sprintf("For vectors of length %d and %d.\n", na, nb)
  }

convolution <- function(a, b)
  {
  ab <- rep(0.0, length(a)+length(b)-1)
  r  <- .C("evaluate", as.numeric(a), as.numeric(b),
                                      as.numeric(ab))
  r[[3]]
```

```
  }

a <- rep(1,6)/6.0
b <- a

lengths(length(a), length(b))
ab <- convolution(a, b)

print(a)
print(b)
dist <- data.frame(values = 2:12, probs = ab)
print(dist)
```

# An Exercise

**Exercise 12.1.** In a *C* program you have variables

```
double a[] = {1, 2, 3, 6, 5, 4};
int m = 2;
int n = 3;
```

but pretend these are the result of long arduous computations. Write interface functions so that in the end you have a $2 \times 3$ matrix

$$\begin{bmatrix} 1 & 2 & 3 \\ 6 & 5 & 4 \end{bmatrix}$$

in *R*. Zip up all your files and email them to me.

# References


[1] Randall L. Eubank and Ana Kupresanin. *Statistical Computing in C++ and R*. The R Series. Chapman & Hall/CRC, 2012.

[2] Norman Matloff. *The Art of* R *Programming*. San Francisco: No Starch Press, 2011.